# The ANL/IBM SP Scheduling System

*David A. Lifka*
Mathematics and Computer Science Division
Argonne National Laboratory
*lifka@mcs.anl.gov*

**Abstract**

During the past five years scientists discovered that modern UNIX workstations connected with ethernet and fiber networks could provide enough computational performance to compete with the supercomputers of the day. As this concept became increasingly popular, the need for distributed queuing and scheduling systems became apparent. Systems such as DQS from Florida State were developed and worked very well. Today, supercomputers, such as Argonne National Laboratory's IBM SP system, can provide more CPU and networking speed than can be obtained from these networks of workstations. These modern supercomputers look like clusters of workstations, however, so developers felt that the scheduling systems that were previously used on clusters of workstations should still apply. After trying to apply some of these scheduling systems to Argonne's SP environment, it became obvious that these two computer environments have very different scheduling needs. Recognizing this need and realizing that no one has addressed it, I developed a new scheduling system. The approach taken in creating this system was unique in that user input and interaction were encouraged throughout the development process. Thus, a scheduler was built that actually "worked" the way the users wanted it to work.

## 1 Background

The Mathematics and Computer Science Division of Argonne National Laboratory acquired a 128-node SP system in order to study parallel computing, scalable I/O, and several other advanced computing areas. The SP system has many types of users, whose various jobs often have conflicting requirements. In order to come up with a "fair" way to schedule these different jobs, several popular scheduling systems were considered. After studying these scheduling systems and actually trying a few, it was determined that none of them could actually suit the needs of our user community. The problem was that these systems had been developed for clusters of high-end workstations connected by fast networks. The authors of these systems had considered all the "best" ways to schedule jobs on such a distributed system, including scheduling I/O-intensive jobs with CPU-intensive jobs, and many other popular, optimistic scheduling schemes. These schedulers can do all sorts of complex things but not the simple things that our users wanted! This situation was quite disturbing, and my task was to find a scheduling system that could satisfy our user community or schedule their jobs by hand round-the-clock! Not being much of a night person,

I opted to write my own scheduling system in which the user community could define its requirements.

## 2 In Search of the Ultimate Scheduler

Before beginning to write a new scheduler, a lot of thought went into what exactly it was a scheduling system should provide. There were three basic goals that almost any scheduling system strives for: fairness, simplicity (easy to understand), and efficient use of the available resources. These three goals are obviously in conflict, so there had to be some compromise that would make the users happy. After a fair amount of research, we developed a list of features that an "Ultimate Scheduler" should:

- Provide optimum performance (e.g., I/O-bound and CPU-bound jobs together)
- Be fair (whatever that means...)
- Support different job classes (e.g., interactive vs. batch)
- Support various message-passing libraries
- Use static or dynamic partitioning of the machine
- Utilize time or space slicing, gang scheduling, or sign-up sheet mechanisms
- Schedule different computation models (task farm vs. parallel processing)
- Manage other system resources (e.g., I/O subsystems)
- Provide priority scheduling for special jobs

Several of these items really depend upon how the users of a machine expect to be able to use it. There are several very nice scheduling systems available today that try to address these issues. A few of the more popular are

- DQS from Florida State
- Condor from University of Wisconsin
- IBM LoadLeveler
- NQS

The problem with these systems is that they all primarily focus on managing multiple queues of non-parallel jobs for networks of workstations. They were developed in the age of the "free supercomputing" movement. This was not too long ago when high-end workstations connected by fast networks could provide as much computational power as the super-computers of the day at a fraction of the cost. Many of these scheduling systems do more than scheduling. Figure 1

shows the main pieces of a complete scheduling system. Several of the available scheduling systems have implemented the various pieces of this diagram in a tightly coupled fashion. This implementation greatly reduces the extensibility of the system. For this reason a scheduling system that would meet the ANL goals addresses only "Scheduling" and attempts to get the other pieces from either the machine vendors or other developers wherever possible.

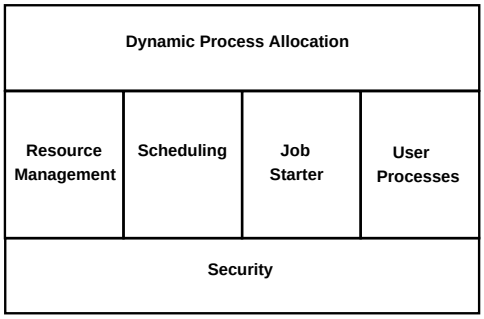


*Figure 1*

# 3 The ANL Scheduler Requirements

Argonne's users and management had their own set of requirements that these systems couldn't quite address. The first was that users had to be able to request a set of nodes for any type of use. ANL users have several different modes of operation. Some need to be able to do task farming, where the SP is used as if it were a large collection of unconnected workstations; others want to run parallel jobs using various message-passing libraries. They need to be able to run jobs interactively and in batch mode. Interactive use allows users to actually log onto the nodes and run their codes by hand. This facilitates debugging and simple use of the machine for less sophisticated users. Batch use allows for large production runs and unattended runs during the night or weekends. Because of these different job types it was important not to statically partition the machine into different-size "pools" of nodes. A 127-node job should be able to run with a 1-node job, as should a 65-node job with a 63-node job. These different jobs have equal importance so the number of requested resources and duration or usage had drive the queuing policy, not the "types" of jobs.

# 4 Addressing Requirements

Several of the members of the Mathematics and Computer Science Division at ANL are researching new message-passing systems, so the scheduler had to be able to make use of any of them. Addressing this requirement was difficult because of a software limitation but led to one of the key concepts of the scheduler.

To use the IBM SP high-performance switch, the users must have exclusive access to it. At the same time, there had to be a fair alternative for SP users who weren't necessarily using the switch, or even doing parallel programming. It turned out the only fair thing to do was to provide any user exclusive access to any number of nodes requested. For several reasons this turned out to be an advantage in the scheduler development. Exclusive access meant that any user would have optimal cache performance, access to all the memory, and access to the full CPU and I/O potential of each node for benchmarking performance.

This "great" idea of exclusive access has a major drawback. If users have exclusive access, what is to keep them from holding the resource and not letting other jobs on the system? There had to be a way to provide exclusive access to the machine and still provide a deterministic run-time for any given job. This is the other key concept in the ANL scheduling system. Users have to provide a run-time in wall-clock node/minutes (like in the days of mainframe computing). Having exclusive access to the nodes allows them to do this, since they will be able to better predict the run-time of their jobs. These key concepts — exclusive access and user-provided run-times — allow for this different approach to scheduling.

There is still one other problem with this idea. What prevents a user from scheduling a job that requires all the resources for a very long time. It quickly became apparent that a new resource-accounting mechanism was needed. Using the system-generated accounting statistics of CPU and I/O usage was not sufficient. Users who "forgot" to use exclusively scheduled time would not be "charged" anything since they consumed no resources. The accounting system had to be based on wall-clock time scheduled, not resources used. When users are given their account, they are given a number of resource-units to use on the machine, in the case of ANL wall-clock minutes. Once they have used all their units, they are not allowed to submit any more jobs to the queue. This effectively prevents users from asking for more time on the machine than they actually need.

# 5 An Attempt at Fairness

Based on the two key scheduler concepts, a FIFO queue was the first queuing method that was implemented. The ANL users ran a variety of jobs on the system. Figure 2 shows the typical resource requirements that were observed.

| Required Nodes | Required Time |
|---|---|
| 1 - 8 nodes | 8 - 48 hours |
| 16 - 32 nodes | 1 - 8 hours |
| 64 - 128 nodes | 30 minutes - 3 hours |

*Figure 2*

Realizing the limitations of a FIFO queue, the scheduler was designed to be very modular so that new or different user requirements could drive the scheduling policy without requiring a complete rewrite of the code. This also provided the capability to plug in different queuing algorithms. Another important difference between the ANL SP Scheduler and others is that users were involved in the development and creating the scheduler policy from the beginning. Rather than try to come up with the optimal computer-science solution, a simple FIFO solution was applied, and users were encouraged to make suggestions for its improvement. To do this was actually simple. Users simply needed to be able to see the current scheduling algorithm and job queue, and to watch the queuing of jobs in operation. This approach allows the users to quickly become acquainted with the problems the scheduler is trying to solve and to suggest improvements in its operation. Having this user interaction allowed the users to help debug the scheduler, and thus its development became a community project.

It quickly became apparent to all that a FIFO queue was extremely inefficient. What typically would happened was that on a 128-node system a job requiring only a few nodes would start and the next job in the queue would require 128 nodes. Hence, a large number of nodes would remain idle until the first job finished and the second job could start. A new scheme was quickly devised. It was dubbed FIFO with backfilling. Backfilling provides a way to fill in the idle nodes caused by the situation just described with other jobs further down the queue, provided that they do not cause the first job in the queue to wait any longer for the nodes they require. Here is an example of a typical queue of jobs and backfilling in action:

Step 1: 128 nodes are idle with the following queue of jobs. User A needs 32 nodes and there are 128 available so it is allowed to start.

| User Name | Number of Nodes | Number of Minutes | Job Status |
|---|---|---|---|
| User A | 32 | 120 | Startable |
| User B | 64 | 60 | Waiting |
| User C | 24 | 180 | Waiting |
| User D | 32 | 120 | Waiting |
| User E | 16 | 120 | Waiting |
| User F | 10 | 480 | Waiting |
| User G | 4 | 30 | Waiting |
| User H | 32 | 120 | Waiting |

Step 2: 96 nodes are idle and 32 are in use with the following queue of jobs. User B needs 64 nodes and there are 64 available so it is allowed to start.

| User Name | Number of Nodes | Number of Minutes | Job Status |
|---|---|---|---|
| User A | 32 | 120 | Running |
| User B | 64 | 60 | Startable |
| User C | 24 | 180 | Waiting |
| User D | 32 | 120 | Waiting |
| User E | 16 | 120 | Waiting |
| User F | 10 | 480 | Waiting |
| User G | 4 | 30 | Waiting |
| User H | 32 | 120 | Waiting |

Step 3: 32 nodes are idle and 96 are in use with the following queue of jobs. User C needs 24 nodes and there are 32 available so it is allowed to start.

| User Name | Number of Nodes | Number of Minutes | Job Status |
|---|---|---|---|
| User A | 32 | 120 | Running |
| User B | 64 | 60 | Running |
| User C | 24 | 180 | Startable |
| User D | 32 | 120 | Waiting |
| User E | 16 | 120 | Waiting |
| User F | 10 | 480 | Waiting |
| User G | 4 | 30 | Waiting |
| User H | 32 | 120 | Waiting |

Step 4: 8 nodes are idle and 120 are in use with the following queue of jobs. User D needs 32 nodes and there are only 8 nodes available so it is not able to start. Now the backfill algorithm has to determine how long User D is blocked, or in other words how long it will be before enough nodes will be available for User D to run. To do this, the scheduler looks at the list of running jobs and determines how long it will be until enough have them have finished for User D to start. User A will be finished in 120 minutes, User B will finished in 60 minutes, and User C 180 minutes. From this list the algorithm determines that when User B finishes in 60 minutes, there will be enough nodes available for User D to start; therefore, User D should have to wait for 60 minutes at the longest. With this information the algorithm now looks at the queue of jobs looking for a job that can use the 8 available nodes for 60 minutes or less. Users E and F require too many nodes so they cannot backfill. User G requires 4 nodes for 30 minutes, which will not delay the start of User D, so it is allowed to start.

| User Name | Number of Nodes | Number of Minutes | Job Status |
|---|---|---|---|
| User A | 32 | 120 | Running |
| User B | 64 | 60 | Running |
| User C | 24 | 180 | Running |
| User D | 32 | 120 | Blocked |
| User E | 16 | 120 | Ineligible |
| User F | 10 | 480 | Ineligible |
| User G | 4 | 30 | Startable |
| User H | 32 | 120 | Waiting |

Now suppose that User F needs 8 nodes instead of 10. Eight nodes are idle and 120 are in use with the following different queue of jobs. User D needs 32 nodes and there are only 8 nodes available, so it is not able to start. Now the backfill algorithm has to determine how long User D is blocked, or in other words how long it will be before enough nodes will be available for User D to run. To do this, it looks at the list of running jobs and determines how long it will be until enough have them have finished for User D to start. User A will be finished in 120 minutes, User B will finished in 60 minutes, and User C 180 minutes. From this list the algorithm determines that when User B finishes in 60 minutes there will be enough nodes available for User D to start; therefore, User D should have to wait for 60 minutes at the longest. With this information the algorithm now looks at the queue of jobs looking for a job that can use the 8 available nodes for 60 minutes or less. User E requires too many nodes, so it cannot backfill. User F requires 8 nodes for 480 minutes, which is longer than the time User D is blocked for; but when User B finishes, it will release 64 nodes, which is more than User D needs. The backfill algorithm determines that there will still be enough nodes for User D to start in 60 minutes if it starts User F, so it is started.

| User Name | Number of Nodes | Number of Minutes | Job Status |
|---|---|---|---|
| User A | 32 | 120 | Running |
| User B | 64 | 60 | Running |
| User C | 24 | 180 | Running |
| User D | 32 | 120 | Blocked |
| User E | 16 | 120 | Ineligible |
| User F | 8 | 480 | Startable |
| User G | 4 | 30 | Waiting |
| User H | 32 | 120 | Waiting |

# 6 Keep It Simple

Another drawback to many of the available scheduling systems is that they can be quite complicated to use and, for naive users, quite intimidating. To avoid this problem, I desgined a minimal set of commands, with functions similar to

the UNIX commands they mimic or to their names. These simple commands can be used to build up more elaborate tools if the users wish to do so. The following list shows the complete set of user commands and a brief explanation of their functionality:

| | |
|---|---|
| sphelp | list user commands and their functions |
| spfree | return the number of free nodes |
| sppause | pause a job waiting in the queue so that it will not be started |
| spunpause | unpause a job waiting in the queue |
| spq | show the jobs currently on the system and waiting in the queue |
| sprelease | release a node back to the free pool |
| spsubmit | submit a job to queue |
| spusage | return a current snap-shot of the resource file |
| spwait | block until a specific job has completed |
| spwhat | return what type of job could be run if submitted now |
| spwhen | tell when a specific job will start given the current queue |
| getjid | return the user job ID on a scheduled node. |

# 7 Summary

The key design features of the ANL SP scheduler are that it provides exclusive access to the nodes the user is allocated and that users provide run-times in wall-clock minutes so that anyone can determine when a job will start. Providing users with enough information to understand the queuing mechanism and the tools to follow its progress in real time allows the users to help in the debugging and enhancement of the scheduler. By using user requirements as design points, a very simple scheduler was able to be developed to satisfy their needs. Although these requirements are seemingly simple, it was surprising to find that many of today's advanced scheduling systems do not support them. More information on the Argonne SP scheduling system and how it addresses IBM SP-specific issues

is available in the *Users Guide to the Argonne SP Scheduling System* [1].

## References


1. Lifka, D., Henderson, M.,and Rayl, K.,ANL/MCS-TM-201, *Users Guide to the Argonne SP Scheduling System*, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL (1995)